

# JS Level 1



# Введение



# События

В этой лекции мы с вами поговорим о событиях – специальном механизме, который позволяет нашей программе получать уведомления о том, что что-то происходит. Например, пользователь кликает мышью.



# Повторение



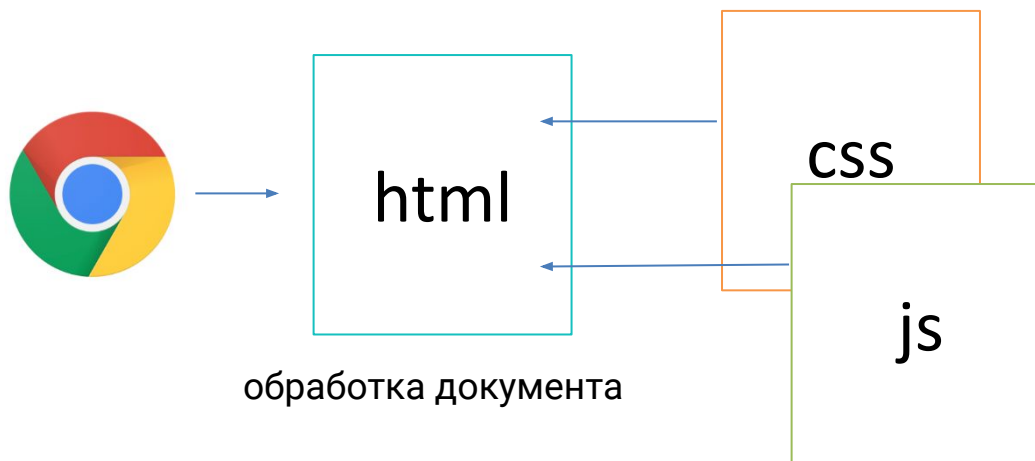
# Web Application

На прошлой лекции мы поговорили с вами, как работают веб-приложения (а именно их клиентская часть) – они загружаются и запускаются в браузере:



# Ресурсы

Кроме того, мы обсудили сам механизм: сначала загружается HTML-документ (если вы указали его в адресной строке), а затем уже все ресурсы, которые в этом самом документе прописаны:



Сегодня наша задача – детально разобраться с тем, как же браузер обрабатывает загруженные ресурсы и что мы можем делать из JS с загруженной страницей.



# Загрузка документа



# Браузер

Всё, что мы будем рассматривать сегодня, мы будем рассматривать на примере браузера Chrome, поскольку он и созданные на базе него браузеры совокупно занимают подавляющую часть рынка.





# Тестовая страница

Начнём мы с того, что создадим типовой проект, но подключать css и js пока не будем:

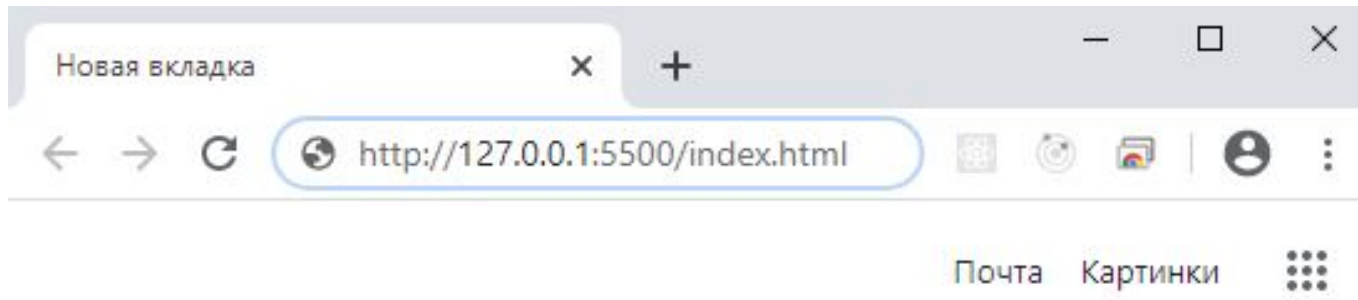
```
<> index.html > ...  
1  <!DOCTYPE html>  
2  <html lang="en">  
3  <head>  
4      <meta charset="UTF-8">  
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">  
6      <title>Document</title>  
7  </head>  
8  <body>  
9      <h1>Hello, JS!</h1>  
10 </body>  
11 </html>
```

Откроем его, как обычно в Live Server, и рассмотрим процесс формирования страницы целиком.



# Загрузка страницы

При открытии через Live Server адрес страницы уже вбит в адресную строку:



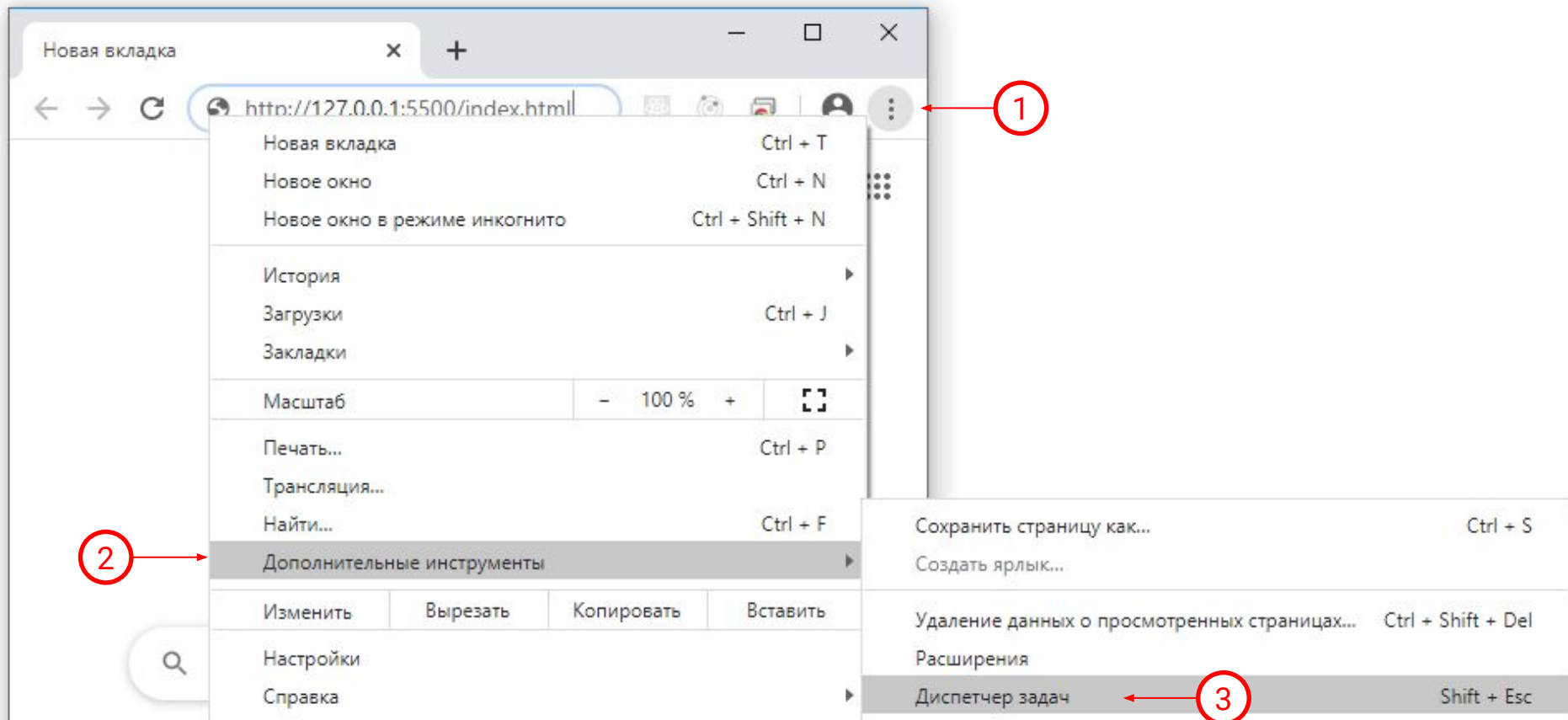
Неважно, был вбит он заранее или вы вбили его руками, происходит следующее: браузер отправляет запрос по этому адресу в надежде загрузить документ.

Как он это делает (т.е. отправляет запрос) мы будем разбирать чуть позже, ключевое это то, что отправляется запрос.



# Процессы

Браузер – не единая программа, а несколько взаимодействующих между собой активностей. Увидеть это можно, следующим образом:



# Процессы

Что такое процесс? Представьте большой магазин, в котором продаются товары. Товары сами себя не продают – нужны специальные люди, которые привозят этот товар, разгружают, выставляют на витрины, прикрепляют ценники.

Кроме того, нужны продавцы на кассах, специальный персонал, который будет взвешивать фрукты и овощи, резать сыр/мясо и т.д.

Нужны ещё охранники и те, кто отвечает за организацию всего.



# Процессы

В компьютере всё устроено так же: пока программа просто лежит в виде файла на вашем диске (например, вы установили VS Code и не запускаете его) – это просто файл, он ничего не делает.

Но когда вы "запустили" его, операционная система, читает весь файл и запускает его на исполнение – создаётся процесс, который уже и может делать работу. В простейшем случае работает схема одна запущенная программа – один процесс. Например, вы можете запустить несколько программ Блокнот (notepad.exe). Сколько раз вы его запустите, столько процессов и создастся.

В сложных программах, таких как браузер – может быть много процессов (это с магазином – в маленьком магазине продавец является и хозяином, и грузчиком, и охранником, а в большом – это разные люди).



# Процессы

Как вы видите, тут есть несколько процессов. Каждый процесс отвечает за свою задачу. Например:

- Браузер – отвечает за отрисовку окна браузера (меню, кнопки вперед назад, закладки и т.д.)
- Network Service – отвечает за отправку и получение данных
- Вкладка: Новая вкладка – отвечает за отображение конкретной вкладки ←



Задача	Объем потребляемой памяти	ЦПУ	Сеть	Идентификатор процесса
 Браузер	594 744K	0.0	0	1572
•  Процесс GPU	507 608K	0.0	0	2044
•  Утилита: Network Service	108 024K	0.0	0	11308
•  Утилита: Audio Service	6 116K	0.0	0	12784
•  Вкладка: Новая вкладка	31 864K	0.0	0	9480



# Процессы

У каждого компьютера есть ограниченный набор ресурсов. Ключевые для нас – это память и процессорное время.

Память – это то, что позволяет программе хранить нужные для неё данные (которые нужны сейчас). Чем больше данных использует сейчас программа, тем больше памяти ей требуется\*.







Примечание\*: конечно же, это очень упрощённое описание, пока нас детали не особо интересуют.



# Процессы

Например, откроем две вкладки:

1. Наш тестовый документ
2. Facebook.com

Задача	Объем потребляемой памяти	ЦПУ	Сеть
 Браузер	559 564K	1.6	0
•  Процесс GPU	609 064K	0.0	0
•  Утилита: Network Service	108 148K	0.0	0
•  Утилита: Audio Service	6 908K	0.0	0
•  Вкладка: Document	20 680K	0.0	0
•  Вкладка: Алиф - Главна...	108 564K	7.8	1 078 КБ/с

В нашем тестовом документе мало информации – потребляется одно количество памяти, а страница Facebook потребляет в разы больше (потому что там есть посты и т.д.)





# Процессы

Что для нас из этого всего важно: важно то, что каждую вкладку обрабатывает отдельный процесс\*. Это процесс называют `Renderer`.

После того, как вы вбиваете адрес страницы в адресную строку, браузер посредством `Network Service` загружает нужные данные и отдаёт `Renderer'у`. Задача `Renderer'a` – превратить набор файлов `HTML`, `CSS` и `JS` в документ, с которым может работать пользователь (скроллить, нажимать на кнопки, выделять текст и т.д.).

Примечание\*: если у вас совсем маломощный компьютер, то `Chrome` может заставить обрабатывать один процесс несколько вкладок (вспомните аналогию про маленький магазин).



# Парсинг

Когда Renderer получает HTML-документ (а если вы помните, то первым загружается именно HTML-документ), он начинает обрабатывать его (говорят "парсить").

Парсинг заключается в том, что ищутся открывающие/закрывающие теги, читается содержимое атрибутов и т.д. При этом, что важно: браузер "исправляет" за вас документ, если видит, что вы его (документ) написали неправильно. Это важно! JS не исправляет, а документ исправляет.

В результате парсинга для каждого элемента создаётся объект (примерно так же, как мы с вами делали на предыдущих лекциях) и браузер "собирает" все эти объекты в определённую иерархию, называемую деревом (мы с вами это обсуждали).



# DOM

Созданное дерево + API для взаимодействия с ним из JS называют DOM (Document Object Model – объектная модель документа\*).

Что это для нас? Для нас – это набор готовых объектов и методов, которые позволяют нам из JS взаимодействовать со страницей (например, менять элементы).

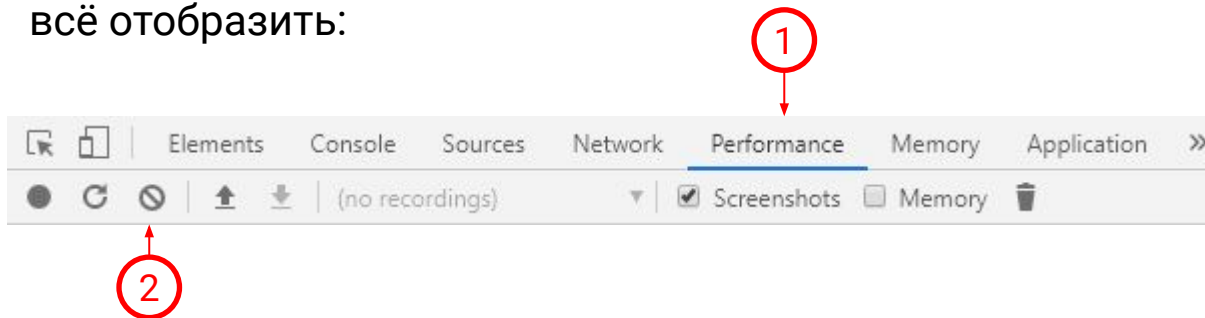
Примечание\*: конечно же, никто её так обычно не называет, все говорят просто DOM.



# Дальнейший процесс

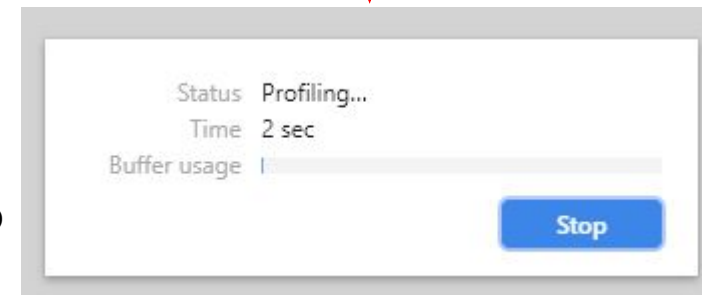
Дальнейший процесс построения страницы нас, на самом деле, пока не очень интересует, но ради интереса, давайте попробуем посмотреть на основные шаги.

Чтобы не быть голословными, давайте посмотрим на то, как Chrome на это может всё отобразить:



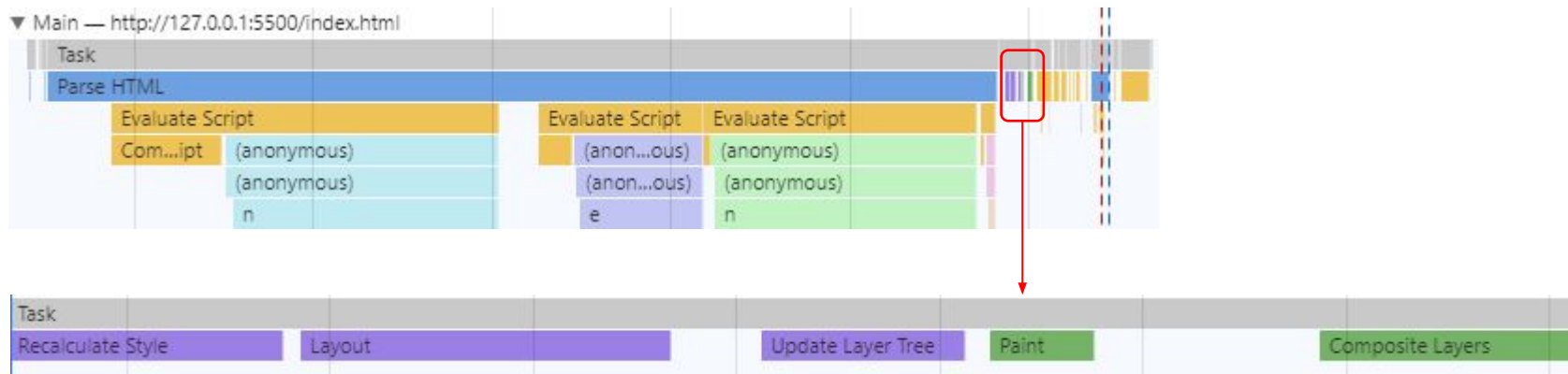
Страница перезагрузится и мы увидим вот такое окошко:

Это значит, что Chrome начал внутри себя замерять, сколько времени он тратит на ту или иную операцию (profiling).



# Дальнейший процесс

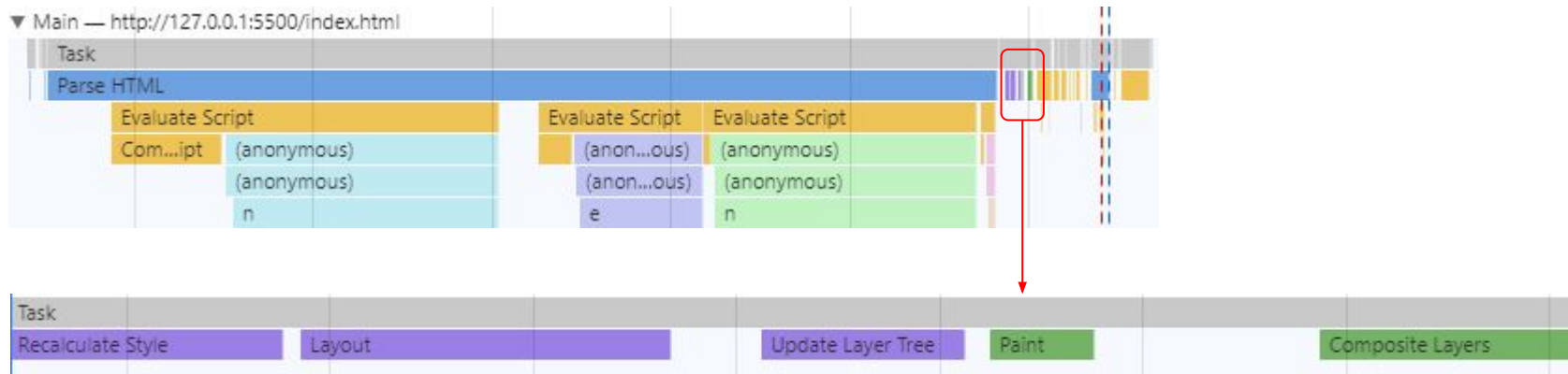
В результате мы получим примерно такую картинку (колёсико мыши позволяет увеличивать масштаб):



Примечание\*: не переживайте, если у вас не получилось с первого раза "управиться" с этим инструментом (нужна практика) – мы его приводим только для того, чтобы вы понимали общий процесс.



# Дальнейший процесс



Как вы видите, большая часть времени ушла именно на парсинг HTML и выполнение скриптов. Но каких скриптов? Мы же ничего не подключаем?

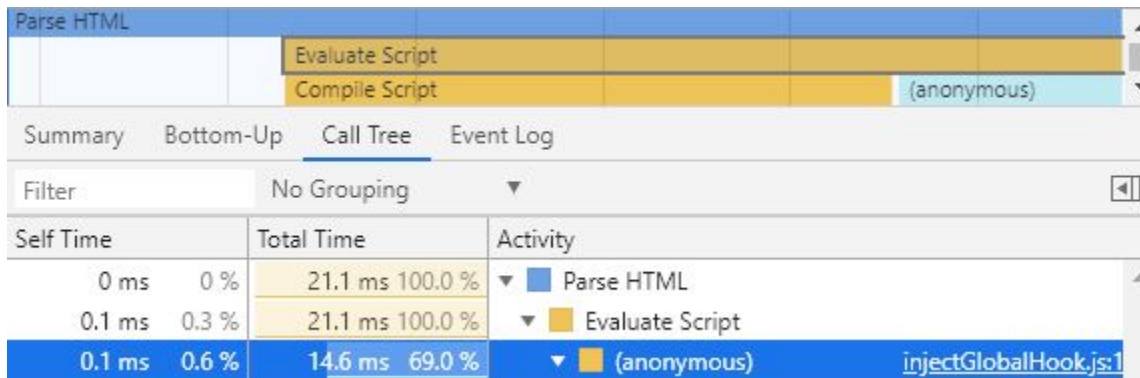
На самом деле, в страничке есть скрипты Live Server + расширения (или дополнения), которые вы ставите в Chrome – это тоже скрипты и они имеют доступ к вашей странице.

Расширения Chrome – это доп.программки, которые вы можете установить через [Chrome Web Store](https://chrome.google.com/webstore/).



# Дальнейший процесс

Например, выделенный кусочек – это выполнение кода расширения React Developer Tools (которое мы будем использовать на курсе по React):



# Дальнейший процесс

После того, как будет построено DOM-дерево, начинается применение стилей (CSS), вычисление размеров и позиционирование элементов, их отрисовка на экране и т.д.



Вы можете себе представлять общий процесс следующим образом: вы хотите поклеить обои в квартире. Конечно же, большинство людей, прежде чем это сделать, как минимум, померяют размеры комнаты и исходя из этого рассчитают сколько рулонов обоев нужно (это и есть вычисление размеров). Потом они примерно определяют, сколько полос на какую стену, как их нужно резать и т.д. (это и есть позиционирование). Ну и отрисовка заключается в непосредственном приклеивании обоев.

Естественно, есть нюансы в этой аналогии, например, многие сначала клеят одну полосу, смотрят "как пошло" и где они ошиблись и т.д.





# Renderer

Запомните про Renderer, т.к. нам это будет нужно сегодня, когда мы с вами будем разговаривать про события.



# Дальнейший процесс

Нас это на данном этапе все эти детали нас не интересуют, но вы можете ознакомиться со статьёй "[How Browsers Work](#)" (ссылка указывает на русскоязычный перевод), в которой описываются детали.

**Важно:** статья достаточно старая (2011 год), поэтому мы её рекомендуем только для общего ознакомления.



# Events



# Event

Когда что-то происходит, браузер создаёт специальный объект, который содержит почти всю информацию о том, что произошло.

Например, пользователь кликает мышью: браузер создаёт объект, в котором информация о том, на каком именно элементе пользователь кликнул, какой кнопкой мыши, были ли зажаты вспомогательные клавиши Control, Shift и т.д.



# Подписка на события

Ключевая идея в следующем: событий происходит огромное количество (каждое наше движение мышью – уже событие), и не все они нам интересны.

Поэтому работает модель "подписки". Мы можем "подписаться" на то, чтобы браузер уведомлял нас о возникновении только конкретных событий на конкретных элементах.



# Обработка событий

Поэтому первое, чему нам нужно научиться – это сообщать браузеру о намерении получать эти самые события.

Для этого создадим новый проект, назовём его Liker. Суть проекта достаточно простая – мы даём пользователю две кнопки, и он может ставить like или dislike нашему посту:



# Разметка

```
11 <body>
12   <div data-type="post">
13     <div>
14       <div>
15         <img data-id="image">
16       </div>
17       <div>
18         ❤️ <span data-id="likes"></span>
19         <button data-action="like">Like</button>
20         <button data-action="dislike">Dislike</button>
21       </div>
22     </div>
23   </div>
24   <script src="js/app.js"></script>
25 </body>
```

Напоминаем, что мы сами придумываем `data`-атрибуты и на этот раз мы решили кнопкам действий присвоить атрибуты `data-action`.



# Поиск элементов

Предыдущая лекция была достаточно насыщенной, поэтому часть вопросов, связанных с правильной работой с DOM, мы вынесли в эту лекцию.

Первый вопрос, который нужно обсудить – это поиск элементов. Суть вопроса заключается в следующем: поиск по всему документу – это достаточно затратная операция, поэтому стараются по максимуму избегать этого (именно

`document.querySelector`):

```
3 > const post = { ...
11 };
12
13 const postEl = document.querySelector('[data-type="post"]');
14 const imgEl = postEl.querySelector('[data-id="image"]');
15 const likesEl = postEl.querySelector('[data-id="likes"]');
16 const likeEl = postEl.querySelector('[data-action="like"]');
17 const dislikeEl = postEl.querySelector('[data-action="dislike"]');
```

ищем в document нужный нам элемент

всё остальное ищем уже внутри этого  
элемента (а не по всему документу)





# Поиск элементов

На небольших проектах мы почти не заметим разницы, а вот на больших (как соц. страницы Vk или, например, страничка с рейтингом на несколько тысяч записей) – уже заметно.



# Поиск элементов

Второй вопрос – это хранение самих данных. Мы вам показали два способа:

1. Данные хранятся отдельно (в ДЗ Тобон)
2. Данные хранятся в свойствах элемента, к которому привязаны (в ДЗ Vk)

Ещё возможно хранить данные в **data**-атрибутах, но очень уж неудобно (т.к. в атрибуты, напоминаем, можно класть только строки).



# Поиск элементов

Предпочтительным является первый способ, поэтому мы всё вынесли в отдельный объект и будем информацию хранить там:

```
1  'use strict';
2
3  const post = {
4    id: 1,
5    title: 'JS is Fun!',
6    img: {
7      url: 'img/logo_js.svg',
8      alt: 'JS Logo',
9    },
10   likes: 0,
11 };
12
13 const postEl = document.querySelector('[data-type="post"]');
14 const imgEl = postEl.querySelector('[data-id="image"]');
15 const likesEl = postEl.querySelector('[data-id="likes"]');
16 const likeEl = postEl.querySelector('[data-action="like"]');
17 const dislikeEl = postEl.querySelector('[data-action="dislike"]');
```



# Разделение

Почему именно так? Потому что на следующей лекции данные нам будут приходить с сервера (именно наши объекты с постами). Это первое, а второе – это из ключевых принципов: данные хранятся отдельно, визуальное представление (элементы) – отдельно.

То же самое мы уже встречали при работе с самого начала:

- HTML определяет структуру
- CSS визуальное представление
- JS логику

Поэтому мы поступим так же.



# Привязка данных

Остаётся только "привязать" наши данные к отображению, можем это сделать в отдельной функции:

```
19 function bindPostToEl(post, el) {  
20   |   // Наш код  
21 }
```

Но возникает вопрос: если поиск по DOM-дереву достаточно дорогая операция, неужели нам придётся передавать все элементы в качестве параметров? А что будет, когда мы будем передавать большой пост, как в соц.сети, где очень много разных полей?



# Привязка данных

Ключевых вариантов два:

1. Мы можем все элементы объединить в один объект
2. Мы можем каждый раз искать, используя `querySelector` и не обращать пока внимания на "дороговизну" операций.

Давайте обсудим их подробнее. Мы остановимся на первом, просто чтобы показать, как это делается.

Но вы должны иметь в виду, что когда вы делаете небольшое приложение, не всегда нужно "заморачиваться" по поводу скорости работы, красоты кода и т.д. Потому что иногда важнее – сделать быстро и проверить, а нужно ли ваше приложение кому-то, будет ли им кто-то пользоваться?



# Преждевременная оптимизация

Когда вы стараетесь всё сразу учитывать, оптимизировать так, чтобы оно работало быстро и при 100 000 элементов на странице (при этом не зная, будет ли у вас хотя бы 100 пользователей) – это называется преждевременная оптимизация. Эта одна из худших черт разработчиков, старайтесь её избегать.

Но вы спросите: почему же тогда мы требуем от вас чистый код, оптимизированные версии и т.д.? Причин всего две:

1. Вы должны понимать как это делается и для чего
2. На собеседовании вас будут спрашивать "как делать правильно", хотя нужно быть готовым, что на работе придётся делать "быстро"

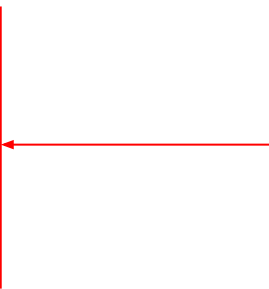


# Виджет

Давайте назовём вновь созданный объект – виджетом и пусть он выглядит вот

```
13  const postEl = document.querySelector('[data-type="post"]');
14  const imgEl = postEl.querySelector('[data-id="image"]');
15  const likesEl = postEl.querySelector('[data-id="likes"]');
16  const likeEl = postEl.querySelector('[data-action="like"]');
17  const dislikeEl = postEl.querySelector('[data-action="dislike"]');
18
19  const postWidget = {
20    rootEl: postEl,
21    imgEl: imgEl,
22    likesEl: likesEl,
23    likeEl: likeEl,
24    dislikeEl: dislikeEl,
25  };

```



**rootEl** – это "корневой" элемент, т.е. тот элемент с которого начинается наш виджет (обычно такие элементы называют рутовыми). Всё остальное – просто свойства.






# Shorthand Properties

Обратите внимание, некоторые свойства пишутся вот так: `imgEl: imgEl`, т.е. имя свойства и имя переменной, в которой хранится значение – одинаковы. Но всё равно приходится печатать по два раза. JS в этом плане проделал большую работу и теперь можно не дублировать это всё:

```
19  const postWidget = {  
20      rootEl: postEl,  
21      imgEl: imgEl,  
22      likesEl: likesEl,  
23      likeEl: likeEl,  
24      dislikeEl: dislikeEl,  
25  };
```

если имя свойства совпадает  
с именем переменной



```
19  const postWidget = {  
20      rootEl: postEl,  
21      imgEl,  
22      likesEl,  
23      likeEl,  
24      dislikeEl,  
25  };
```

Называется это Shorthand Properties.

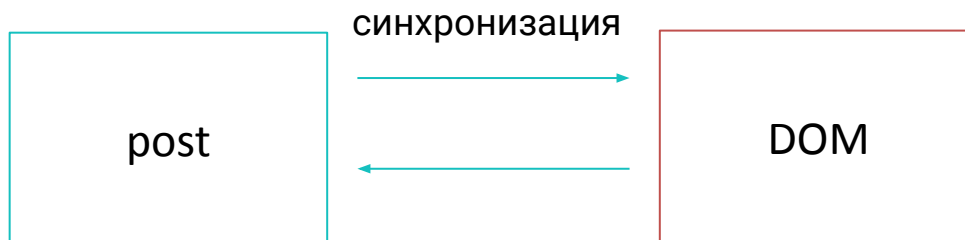
Теперь осталось только дописать функцию, которая связывает объект и виджет.



# Binding

```
27 function bindPostToEl(post, el) {  
28     el.imgEl.src = post.img.url;  
29     el.imgEl.alt = post.img.alt;  
30     el.likesEl.textContent = post.likes;  
31 }  
32  
33 bindPostToEl(post, postWidget);
```

Становится немного сложно, правда? На самом деле, при работе с DOM самое сложное – это постоянно выполнять операцию синхронизации между тем объектом, что хранится в памяти и тем, что сейчас находится в DOM-дереве:



Именно поэтому появились фреймворки Angular, React и другие, которые это делают за вас. Но для начала нужно попробовать обойтись без них.

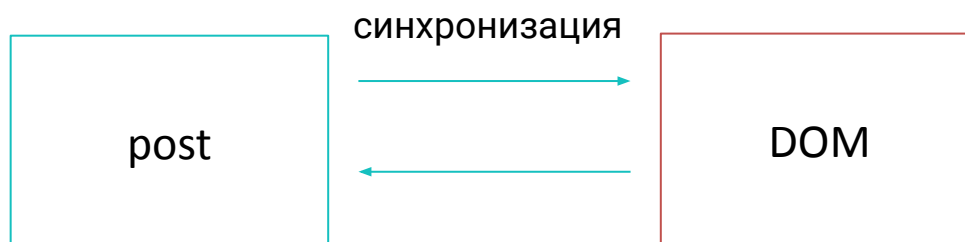


# Binding

```
27 function bindPostToEl(post, el) {  
28     el.imgEl.src = post.img.url;  
29     el.imgEl.alt = post.img.alt;  
30     el.likesEl.textContent = post.likes;  
31 }  
32  
33 bindPostToEl(post, postWidget);
```

Именно в этом суть нашей функции `bindPostToEl` – она "обновляет" данные в DOM-дереве (а именно свойства элементов) в соответствии с тем, что сейчас хранится в `post`'е.

Самое главное: мы можем вызывать эту функцию каждый раз, когда что-то произойдёт с нашим постом и тогда DOM-дерево обновится:



# Event Handler'ы



# Event Handler'ы

Первая возможность подписаться на события – это Event Handler'ы (или обработчики событий). Если мы посмотрим на любой элемент, то в нём окажется много свойств с префиксом **on\***:

```
> dir($(' [data-action="like"] '));
```

```
▼ button ⓘ
```

```
  accessKey: ""
```

```
  ...
```

```
  onclick: null
```

```
  onclose: null
```

```
  oncontextmenu: null
```

```
  oncopy: null
```

```
  oncuechange: null
```

```
  oncut: null
```

```
  ondblclick: null
```

```
  ondrag: null
```

```
  ondragend: null
```

```
  ondragenter: null
```

```
  ondragleave: null
```

```
  ondragover: null
```

```
  ondragstart: null
```

```
  ondrop: null
```

```
  ...
```

в консоли неудобно писать `document.querySelector` – поэтому его можно заменить `$`, а `querySelectorAll` – `$$`



# Event Handlers

Эти свойства описаны в спецификации HTML по названием [Event Handler'ы](#):

<a href="#">Event handler</a>	<a href="#">Event handler event type</a>
<code>onabort</code>	<code>abort</code>
<code>onauxclick</code>	<code>auxclick</code>
<code>oncancel</code>	<code>cancel</code>
<code>oncanplay</code>	<code>canplay</code>
<code>oncanplaythrough</code>	<code>canplaythrough</code>
<code>onchange</code>	<code>change</code>
<code>onclick</code>	<code>click</code>
<code>onclose</code>	<code>close</code>
<code>oncontextmenu</code>	<code>contextmenu</code>
<code>oncuechange</code>	<code>cuechange</code>
<code>ondblclick</code>	<code>dblclick</code>

Выберем "`click`" попадём на длинное описание и прочее-прочее. Это всё хорошо, но что же делать нам?



# Event Handler'ы

На самом деле, всё достаточно просто, нам достаточно в нужное свойство положить функцию, которая будет вызываться браузером при возникновении события:

```
19 > const postWidget = { ...
25   };
26
27   postWidget.likeEl.onclick = evt => {
28     console.log('like clicked');
29     console.log(evt);
30   };
31
32   postWidget.dislikeEl.onclick = evt => {
33     console.log('dislike clicked');
34     console.log(evt);
35   };
36
37 > function bindPostToEl(post, el) { ...
41 }
```

мы свернули ряд объявлений,  
чтобы было больше места

назначаем "обработчики"



# Event Handler'ы

Теперь попробуем кликнуть. Что получится? Всё верно, запустится наш код (вид панельки Console из DevTools):

```
like clicked
```

```
▶ MouseEvent {isTrusted: true, screenX: 138, screenY: 213, clientX: 65, clientY: 95, ...}
```

```
>
```





# Event

То, что печатается в консоли, и есть объект события:

```
▼ MouseEvent {isTrusted: true, screenX: 138, screenY: 213, clientX: 65, clientY: 95, ...} ⓘ  
  altKey: false  
  bubbles: true  
  button: 0  
  buttons: 0  
  cancelBubble: false  
  cancelable: true  
  clientX: 65  
  clientY: 95  
  composed: true  
  ctrlKey: false  
  currentTarget: null  
  defaultPrevented: false  
  detail: 1  
  eventPhase: 0  
  fromElement: null  
  isTrusted: true
```

Информацию об этом событии мы сможем найти в MDN или в спецификации. Пока же нам особо это не нужно.



# Event Handler

Обратите внимание: вам очень важно понять эту модель. Не мы спрашиваем браузер о том, произошло событие или нет, а он уведомляет нас.

Это как в жизни: можно каждые 5 минут звонить человеку и спрашивать, произошло событие или нет (прилетел ли самолёт), а можно оставить свой телефон и попросить перезвонить.

В веб выбран второй путь по одной простой причине: бессмысленно спрашивать про то, нажал пользователь кнопку или нет, потому что он может её вообще никогда не нажать. И только когда он её нажмёт, вот тогда сработает наша функция.



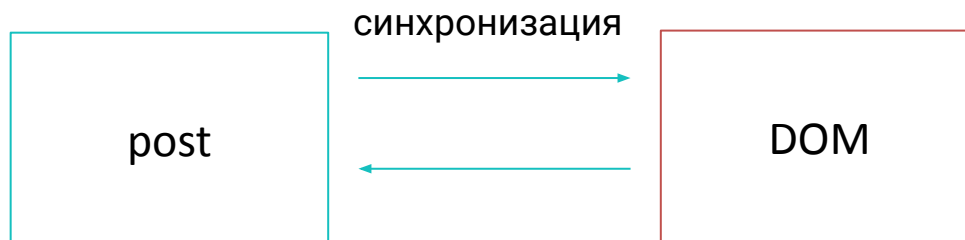
# Event Handler

Кроме того, вы должны запомнить, что не мы вызываем эту функцию, её вызывает сам браузер и он же туда кладёт объект события первым параметром. Мы сокращённо будем называть его `evt` (от `event`). В разных источниках вы можете встретить разные стили именования, но самые частые: `e` (плохой вариант), `ev`, `evt`, `event`.



# Event Handler'ы

Теперь осталось совсем немного: нужно увеличить количество лайков в объекте `post`'а и обновить отображаемое значение в DOM. Напоминаем, DOM автоматически не обновляется – он обновляется только если вы сами его обновите (вызовите установку свойства):



И как раз для этого мы с вами написали отдельную функцию.



# Like & Dislike

```
27 postWidget.likeEl.onclick = evt => {  
28     post.likes++;  
29     bindPostToEl(post, postWidget);  
30 };  
31  
32 postWidget.dislikeEl.onclick = evt => {  
33     post.likes--;  
34     bindPostToEl(post, postWidget);  
35 };
```

Очень интересно. По крайней мере, несколько моментов:

1. Как работают операторы `++` и `--`?
2. Насколько правильно вызывать функцию `bindPostToEl`, ведь она заново устанавливает все свойства?
3. А с именами `post` и `postWidget` всё гораздо интереснее – мы их не объявляли в нашей функции, но при этом всё работает



# ++ И --

Операторы `++` и `--` (называются инкремент и декремент, соответственно) занимаются тем, что увеличивают/уменьшают результат на единицу и сохраняют обратно.

При этом если они пишутся после имени переменной/свойства (например, `post.likes++`), то называются постфиксными и работают следующим образом:

1. Значение в `post.likes` увеличивается на единицу
2. На место выражения подставляется предыдущее (то, которое было до увеличения) значение

Если же пишутся до имени переменной/свойства (например, `++post.likes`), то называются префиксными и работают следующим образом:

1. Значение в `post.likes` увеличивается на единицу
2. На место выражения подставляется текущее (то, которое стало после увеличения) значение



# ++ и --

С постфиксными достаточно легко ошибиться, если их использовать в выражениях. Пример:

```
// в post.likes было 0
```

```
const value = post.likes++;
```

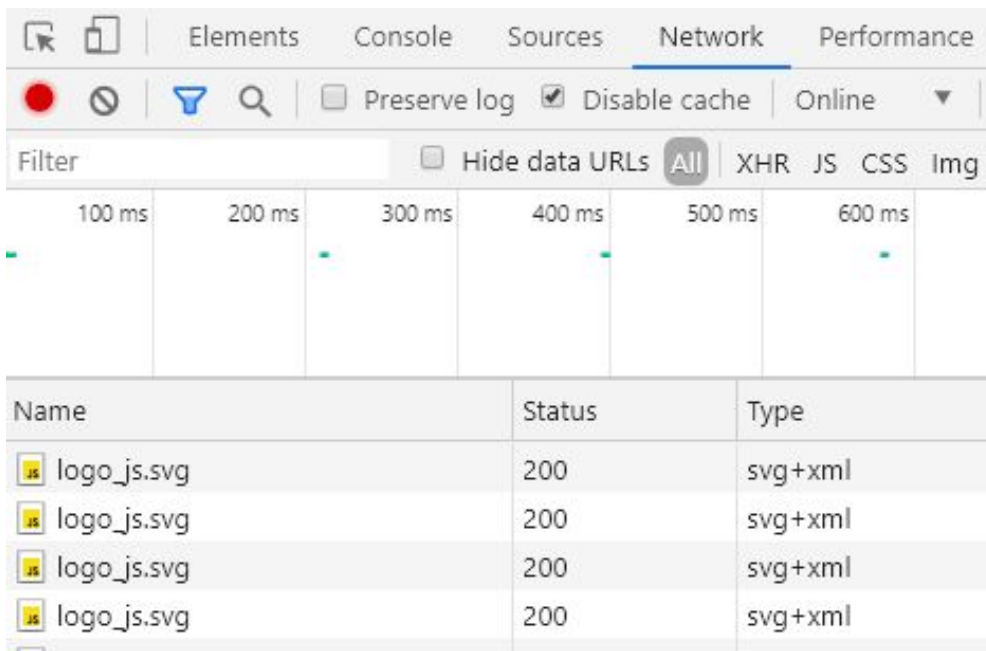
```
// в value будет 0
```

Но если вы их используете без выражений, т.е. просто `post.likes++` (как мы), то никаких проблем не будет.



# Кэширование

С операторами ++ и -- мы разобрались, давайте обсудим ситуацию с `bindPostToEl`. Если у вас во время разработки открыт DevTools (а он должен быть открыт, если вы собираетесь программировать), то перейдя на вкладку Network вы заметите, что при каждом клике браузер заново "скачивает" картинку (и если у вас лимитированный трафик, а картинку вы храните не локально, а берёте из сети Интернет, то вы так "накачаете" много лишнего):





# Кэширование

Что происходит? Дело в том, что мы устанавливаем атрибуты элементов среди которых есть ресурсы. Поэтому браузеру нужно эти ресурсы "перекачать" и заново отрисовать.

Но браузер заботится о пользователях и если он видит, что он уже скачивал запрос по такому адресу и где-то его сохранил\*, он может его не запрашивать. В режиме DevTools браузер запрашивает заново все ресурсы, если включен флажок "Disable Cache":

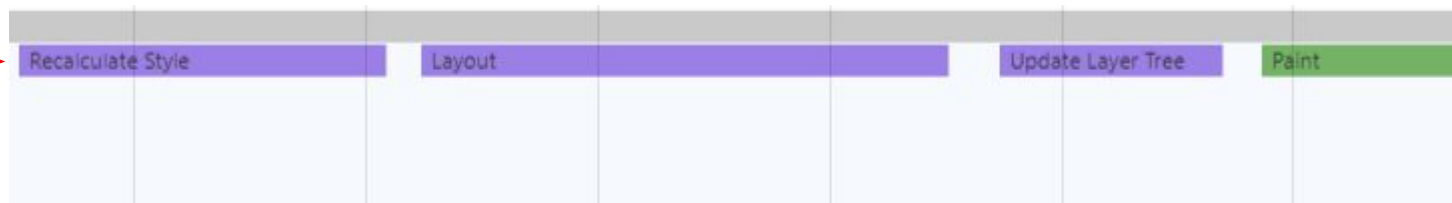
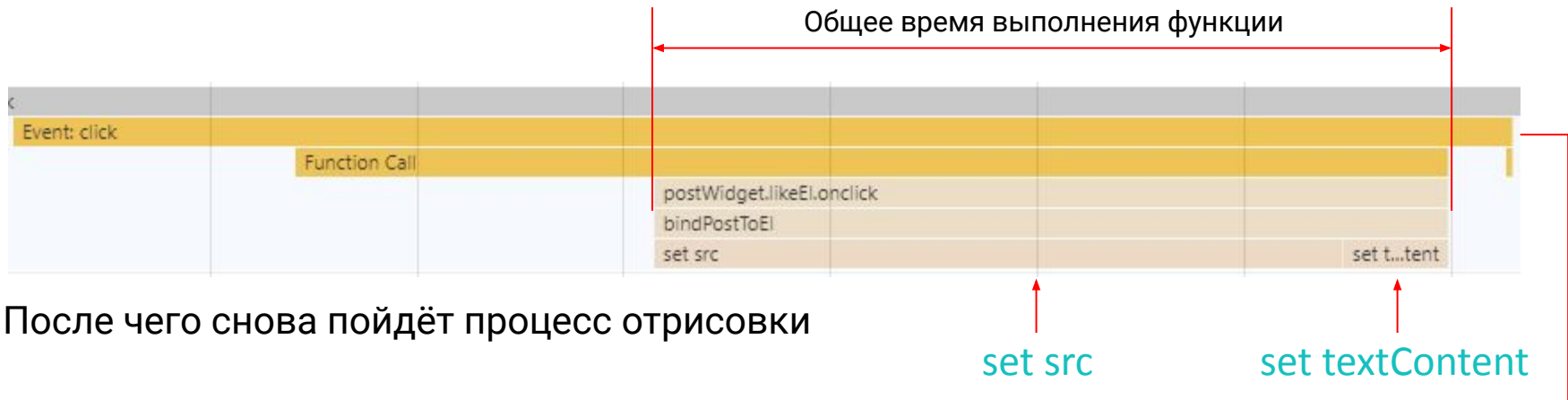


Отключите его и картинка не будет скачиваться при каждой установке свойства [src](#). Примечание\*: про кэширование мы будем детально с вами говорить на следующих курсах, когда будем писать серверную часть.



# Renderer

Для информации (вам этого делать не нужно), на вкладке Performance можно записать активность во время клика и увидеть, что будет делать браузер:



На самом деле, наша функция устроена не очень хорошо, т.к. каждый раз обновляет все атрибуты, хотя можно было посмотреть, что поменялось и обновить только это. Но об этом мы поговорим, когда будем изучать React.




# Closures



# Замыкания

Возвращаемся к вопросу про `bindPostToEl`. Откуда функция взяла имена, которые не объявлены внутри неё?

```
27 postWidget.likeEl.onclick = evt => {  
28   post.likes++;  
29   bindPostToEl(post, postWidget);  
30 };
```



Если в функции используются какие-то имена, которые внутри неё не объявлены, то при запуске функции эти имена ищутся снаружи.

Простая аналогия: сидите вы за компьютером и вдруг вспомнили, что вам нужно позвонить. Вы сначала начинаете искать телефон на рабочем столе (область внутри функции), если не находите – начинаете искать в других комнатах (снаружи). Аналога глобального объекта в данном примере нет.



# Замыкания

В JS функция может обращаться к именам, которые внутри неё не объявлены.

Значения она получает из имён **в момент вызова**.

Проведём небольшой эксперимент, чтобы вам было понятнее (поменяем местами объявление `post`'а и event handler'а):

Всё будет по-прежнему работать. Но как?

```
17 postWidget.likeEl.onclick = evt => {
18   post.likes++;
19   bindPostToEl(post, postWidget);
20 };
21
22 postWidget.dislikeEl.onclick = evt => {
23   post.likes--;
24   bindPostToEl(post, postWidget);
25 };
26
27 const post = {
28   id: 1,
29   title: 'JS is Fun!',
30   img: {
31     url: './img/logo_js.svg',
32     alt: 'JS Logo',
33   },
34   likes: 0,
35 };
```

# Замыкания

Давайте вспомним, первые лекции: браузер загружает наш js-файл и выполняет сверху вниз, не заходя в функции (зелёная полоска сбоку):

```
17 postWidget.likeEl.onclick = evt => {
18   post.likes++;
19   bindPostToEl(post, postWidget);
20 };
21
22 postWidget.dislikeEl.onclick = evt => {
23   post.likes--;
24   bindPostToEl(post, postWidget);
25 };
26
27 const post = {
28   id: 1,
29   title: 'JS is Fun!',
30   img: {
31     url: './img/logo_js.svg',
32     alt: 'JS Logo',
33   },
34   likes: 0,
35 };
```

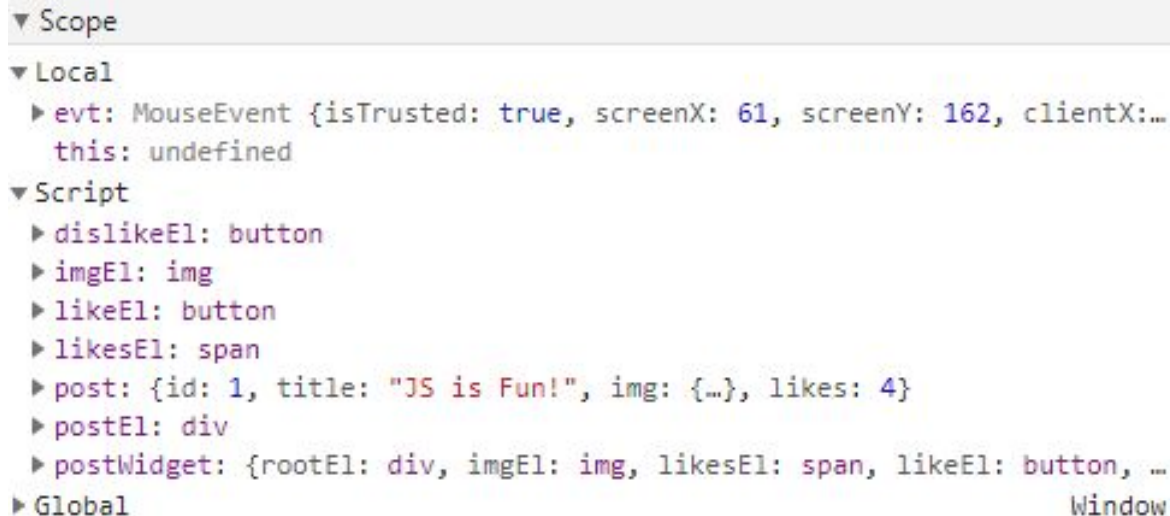
Поскольку он этот код уже выполнил, то имя `post` создалось, хотя функция-обработчик не вызывалась.

И теперь, когда она будет выполняться, она будет искать "ближайшее" существующее имя за своими пределами (поскольку внутри функции такого имени не объявлено).



# Замыкания

Увидеть это всё можно, поставив точку остановки внутри event handler'a:



```
▼ Scope
▼ Local
  ▶ evt: MouseEvent {isTrusted: true, screenX: 61, screenY: 162, clientX:...
    this: undefined
▼ Script
  ▶ dislikeEl: button
  ▶ imgEl: img
  ▶ likeEl: button
  ▶ likesEl: span
  ▶ post: {id: 1, title: "JS is Fun!", img: {...}, likes: 4}
  ▶ postEl: div
  ▶ postWidget: {rootEl: div, imgEl: img, likesEl: span, likeEl: button, ...
  ▶ Global
    Window
```

порядок поиска имён

- Local – значит то, что объявлено внутри функции (или блока, если он есть)
- Script – значит то, что объявлено на верхнем уровне скрипта (вне функций)
- Global – вы уже знаете



# Замыкания

Но тогда возникает вопрос: а зачем мы вообще передаём в функции параметры, если можно использовать внешние имена? Например вот здесь:

```
42 function bindPostToEl(post, el) {  
43     el.imgEl.src = post.img.url;  
44     el.imgEl.alt = post.img.alt;  
45     el.likesEl.textContent = post.likes;  
46 }  
47  
48 bindPostToEl(post, postWidget);
```

Внешние имена нужно использовать только тогда, когда у вас нет других вариантов (например, поскольку event handler вызываем не мы, то мы не можем передать в него параметры, поэтому приходится их использовать).

Использование этого подхода в больших количествах ведёт к тому, что ваши функции буквально гвоздями прибиваются к той точке, где они объявлены и их потом очень сложно менять и модернизировать.





# Замыкания

Если давать определение этой возможности – это называется замыкание (closure).

Closure – это функция, "запоминающая" своё лексическое окружение, в котором была создана, и которая может использовать это окружение в момент вызова, вне зависимости от места вызова (звучит сложно, но по-простому, функция может обращаться к именам вне себя в момент своего вызова)



# Типы событий



# Типы событий

Хорошо, мы с вами посмотрели на событие клика, какие ещё события бывают?

Можно, как всегда, посмотреть либо в спецификации, либо в MDN:

<https://developer.mozilla.org/ru/docs/Web/Events>

Самые популярные (в порядке убывания):

- click
- события формы (познакомимся на следующей неделе)
- value change (познакомимся на следующей неделе)
- события ресурса и прогресса (познакомимся на следующей неделе)

Дальше уже идут специфичные: например, нам понадобятся медиа-события, поскольку какая может быть социальная сеть без загрузки видео и аудио?



# Поведение по умолчанию

Ряд событий имеет поведение по умолчанию: например, при клике по ссылке происходит загрузка той страницы, на которую ведёт ссылка (если она ведёт на новую страницу). В рамках конкретного события описывается и поведение по умолчанию и возможность отменить его (именно для этого нам и нужен в аргументах event handler'a объект события):

Type	click
Interface	<u>PointerEvent</u>
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element
Cancelable	Yes
Composed	Yes
Default action	Varies

← можно ли отменять поведение по умолчанию

← поведение по умолчанию

Для нас это будет важно при обработке форм. Пока же вам нужно обращать внимание на два эти поля при просмотре типов событий.



# Удаление обработчика



# Удаление обработчика

Удаление обработчиков – достаточно редкая операция. Но мы для полноты картины её рассмотрим.

В чём суть: представим, что мы разрешаем пользователю только один раз нажимать на **like** или **dislike** (суровая социальная сеть: обычно пользователю дают возможность как "выразить своё отношение", так и отменить это действие).

В случае event handler'ов всё достаточно просто: когда event handler'a не было, то в свойстве **onclick** было записано значение **null**.



# Удаление обработчика

Поэтому всё удаление сведётся к следующему коду:

```
17 postWidget.likeEl.onclick = evt => {  
18     post.likes++;  
19     postWidget.likeEl.onclick = null;  
20     postWidget.dislikeEl.onclick = null;  
21     bindPostToEl(post, postWidget);  
22 };  
23  
24 postWidget.dislikeEl.onclick = evt => {  
25     post.likes--;  
26     postWidget.likeEl.onclick = null;  
27     postWidget.dislikeEl.onclick = null;  
28     bindPostToEl(post, postWidget);  
29 };
```

Код дублируется, желательно вынести  
в отдельную функцию и вызывать её

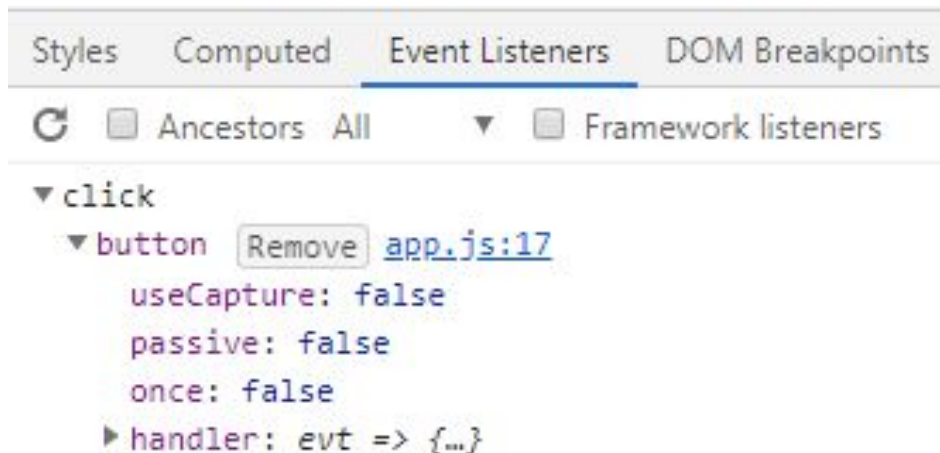
Теперь повторные нажатия на кнопки ни к чему не приведут. Это не совсем хорошо, по одной простой причине: пользователь не будет понимать что происходит (кнопки-то активны) и будет либо дальше кликать, либо пытаться обновлять страничку.



# Удаление обработчика

Как посмотреть, что обработчики действительно есть и что они удаляются?

У нас есть панелька Elements, в которой можно выбрать элемент и на вкладке Event Listeners увидеть все назначенные обработчики событий (и даже удалить их):





# Альтернативы

Давайте посмотрим на альтернативы удаления обработчиков. Чаще всего выбирают один из двух вариантов:

1. "Отключают" кнопки (тоже не очень хороший вариант)
2. Выводят пользователю сообщение

Давайте посмотрим на оба варианта.



# Отключаем кнопки

За включенность или отключенность кнопок отвечает атрибут `disabled`. Когда он пишется в HTML, то его значение – пустая строка. Когда же мы получаем доступ к нему через свойства, то он имеет тип `boolean` (это то, о чём мы говорили в предыдущей лекции, когда разговаривали про свойства и атрибуты):

```
17 postWidget.likeEl.onclick = evt => {  
18   post.likes++;  
19   postWidget.likeEl.disabled = true;  
20   postWidget.dislikeEl.disabled = true;  
21   bindPostToEl(post, postWidget);  
22 };  
23  
24 postWidget.dislikeEl.onclick = evt => {  
25   post.likes--;  
26   postWidget.likeEl.disabled = true;  
27   postWidget.dislikeEl.disabled = true;  
28   bindPostToEl(post, postWidget);  
29 };
```

Обратите внимание, что браузер достаточно умный и после того, как мы отключили кнопки, уже не вызывает event handler'ы (т.к. считает, что раз они отключены, то на них не должны срабатывать обработчики).

Этот вариант плох тем, что пользователь не понимает, почему кнопки отключены и начинает в них кликать активнее.



# Продвинутые юзеры

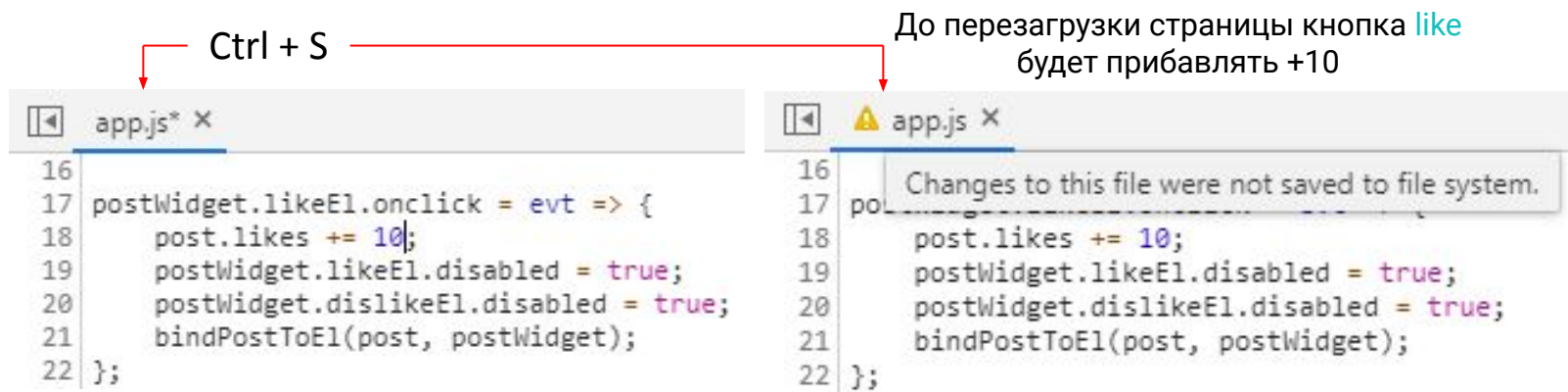
Запомните, что вы не один (одна) такой (такая) программист. Любой пользователь может открыть вкладку Elements и спокойно оттуда убрать атрибут `disabled` (и кнопка снова включится и обработчики снова начнут работать):

```
<span data-id="likes">1</span>
<button data-action="like" disabled>Like
</button> == $0
<button data-action="dislike" disabled>Dislike
</button>
```



# Продвинутые юзеры

Поэтому, как фронтенд-разработчик, запомните: вы отдали страницу и свои скрипты в браузер пользователя и он там может делать всё, что хочет, вплоть до того, что прямо во вкладке Sources редактировать ваши скрипты:



# Юным хакерам

Не используйте эти приёмы на чужих сайтах. В зависимости от страны, в которой расположен веб-сайт, это может караться вплоть до уголовного наказания.



# Сообщение пользователю

Это лучший вариант, который позволит пользователю понять, что что-то пошло не так.

Для этого нам придётся немного поменять разметку:

```
11 <body>
12   <div data-type="post">
13     <div>
14       <div>
15         <img data-id="image">
16       </div>
17       <div>
18         ❤️ <span data-id="likes"></span>
19         <button data-action="like">Like</button>
20         <button data-action="dislike">Dislike</button>
21       </div>
22       <div data-id="message"></div>
23     </div>
24   </div>
25   <script src="js/app.js"></script>
26 </body>
```



# Сообщение пользователю

И добавить в сам post свойство, которое будет отвечать за то, что пользователь уже поставил лайк или дизлайк (а кроме того, добавить в свойства виджета сам элемент):

```
8  const messageEl = postEl.querySelector('[data-id="message"]');
9
10 const postWidget = {
11   rootEl: postEl,
12   imgEl,
13   likesEl,
14   likeEl,
15   dislikeEl,
16   messageEl,
17 };
18
19 const post = {
20   id: 1,
21   title: 'JS is Fun!',
22   img: {
23     url: './img/logo_js.svg',
24     alt: 'JS Logo',
25   },
26   likes: 0,
27   voted: false,
28 };
```



# Сообщение пользователю

```
30 postWidget.likeEl.onclick = evt => {
31     if (post.voted) {
32         postWidget.messageEl.textContent = 'You already voted!';
33         return;
34     }
35     post.likes++;
36     post.voted = true;
37     bindPostToEl(post, postWidget);
38 };
39
40 postWidget.dislikeEl.onclick = evt => {
41     if (post.voted) {
42         postWidget.messageEl.textContent = 'You already voted!';
43         return;
44     }
45     post.likes--;
46     post.voted = true;
47     bindPostToEl(post, postWidget);
48 };
```



else не нужен, достаточно if + return





# Сообщение

Если убрать дублирование кода и добавить стилей + убирать сообщение через определённое время (привет, `setTimeout`), то получится вполне неплохо.

Как вы заметили, вариант с сообщением оказался самым трудоёмким – это нормально. Перед вами всегда будет стоять выбор: либо сделать всё удобно для пользователя (а там будет куча мелочей даже со скрытием сообщения по таймауту), либо сделать быстро.

Решать нужно исходя из ситуации – если у вас запуск проекта завтра, то нужно делать быстро и "чтобы работало", а если есть время – то делать удобно и с проработкой всех мелочей.



# Freezing



# Freezing

Мы достаточно долго рассказывали вам про Renderer и вот пришла пора ответить на вопрос "зачем"?

Проведём следующий эксперимент: вставим "тяжёлый код" в один из обработчиков. Сам код нам не интересен: всё что он делает – это увеличивает значение переменной на 1 несколько миллиардов раз:

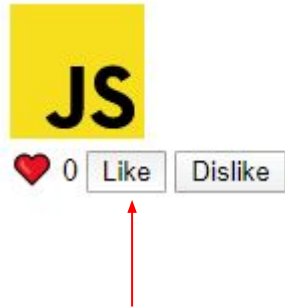
```
30 postWidget.likeEl.onclick = evt => {  
31   if (post.voted) {  
32     postWidget.messageEl.textContent = 'You already voted!';  
33     return;  
34   }  
35   let counter = 0;  
36   for (let i = 0; i < 100_000_000_000; i++) {  
37     counter++;  
38   }  
39  
40   post.likes++;  
41   post.voted = true;  
42   bindPostToEl(post, postWidget);  
43 };
```

← "бессмысленный тяжёлый код"



# Freezing

А теперь попробуем кликнуть (не делайте этого, если у вас слабый компьютер – может зависнуть и придётся перезагружать):



кнопка "зажалась"  
и не отжимается  
все повторные нажатия  
игнорируются\*

Примечание\*: не совсем правда

Почему так произошло? Вся работа нашего скрипта происходит в `Renderer`'е, а значит, пока мы не досчитаем, он не может перерисовать элементы.

Такое поведение называется `freezing` (замораживание). Поэтому первое правило, которое есть в обработке событий и вообще в JS – не делать долгих и тяжёлых вычислений в `Renderer`'е. Есть трюки, позволяющие разбить вычисления на кусочки либо вынести из `Renderer`'а (но об этом позже).



# Итоги



# ИТОГИ

В этой лекции мы обсудили самую важную тему, которая позволяет вам организовать взаимодействие с пользователями.

Мы обсудили только малую её часть, рассказав вам про Event Handler'ы. Помимо Event Handler'ов есть ещё Event Listener'ы – это возможность добавлять более одного обработчика одного и того же события на одном элементе. Кроме того, остаётся тема того, как событие "путешествует" по документу.

Этому всему будет посвящена наша следующая лекция.



# Домашнее задание



# Орг.моменты

Практикум состоит из 8 обязательных занятий. Мы выкладываем новые занятия каждый понедельник в 14:00 (по Душанбе), кроме первой недели.

**Каждое воскресенье в 23:59 (по Душанбе) дедлайн** сдачи домашнего задания. Дедлайн – это предельный срок, до которого вы должны сдать ДЗ.

Если не успеете сдать в срок домашнее задание, тогда этот практикум будет для вас закончен и вы сможете зарегистрироваться на запуск следующего через несколько месяцев.

Все вопросы вы сможете задавать в [Телеграм канале](#).

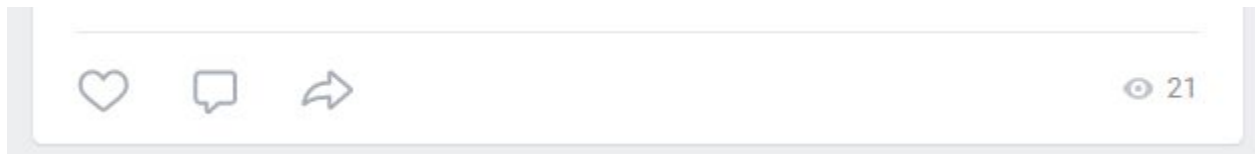




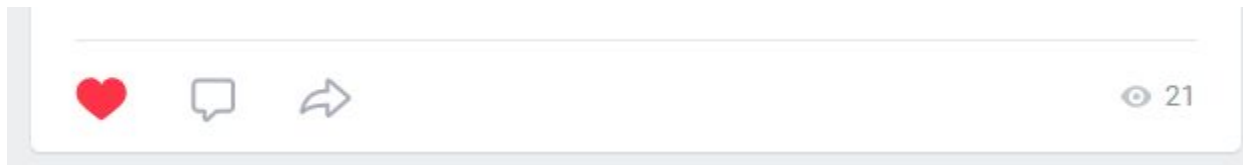
# ДЗ №1: Liker

Мы на лекции сделали неплохой лайкер, а вы сделаете красивый. У постов в Vk в самом низу есть панелька, на которой можно поставить лайк (дизлайк поставить нельзя).

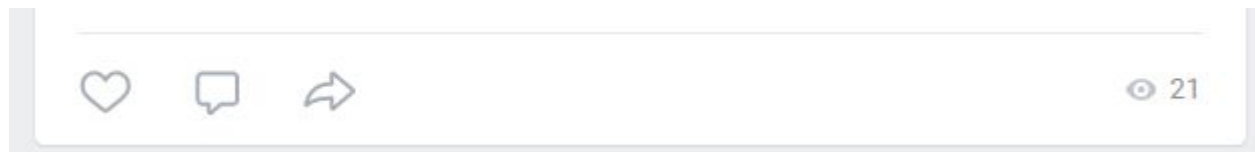
Как это работает: вы заходите, видите такую картинку:



Нажимаете на иконку сердечка, оно становится красным:





Нажимаете ещё раз – становится обратно серым:



# ДЗ №1: Liker

Что нужно сделать:

1. Создайте переменную `post`, в которой должны быть поля `id` (число) и `userLiked` (`boolean`) – остальные по вашему желанию
2. В посте должен быть элемент с `data-action="like"`, внутри которого картинка. Если пост залайкан, то у картинки в `src` находится вот эта картинка:  , если нет, то вот эта:  (сохраните картинки локально и указывайте адреса как [img/liked.svg](#) и [img/unliked.svg](#)\*)

Бот будет открывать вашу страничку, искать пост и лайкать/дизлайкать его (при первой загрузке пост не должен быть залайканным).

Примечание\*: можно, конечно, через CSS менять цвет, но пока нам хватит этой реализации.



# ДЗ №1: Liker

Код в архиве должен располагаться в каталоге [liker](#).



# ДЗ №2: Photo Viewer

Мы решили поместить в нашу соц.сеть "viewer" фотографий, аналогичный тому, то есть в Vk:



# ДЗ №2: Photo Viewer

Бота устроит следующая разметка:

```
25 <body>
26   <div data-id="viewer">
27     <div>
28       <img data-id="photo" src="" alt="">
29     </div>
30     <div>
31       <button data-action="prev">◀</button>
32       <button data-action="next">▶</button>
33     </div>
34   </div>
35 </body>
```

Конечно же, не забудьте подключить скрипт.

По нажатию на кнопки предыдущее и следующее у картинки с `data-id="photo"` меняется атрибут `src` и `alt`.



# ДЗ №2: Photo Viewer

Картинки должны храниться в массиве photos:

```
const photos = [  
  { id: 1, src: '...', alt: '...', },  
  { id: 3, src: '...', alt: '...', },  
  { id: 2, src: '...', alt: '...', },  
  { id: 4, src: '...', alt: '...', },  
];  
  
function bindPhotoToViewer(photo, el) {  
  // ваш код  
}
```

Картинок должно быть всего 4 и должна быть функция `bindPhotoToViewer`.



# ДЗ №2: Photo Viewer

Как всё должно работать:

1. При загрузке страницы в Viewer загружается первое фото, кнопка "предыдущее" отключена
2. При нажатии на "следующее" загружается второе фото, кнопка "предыдущее" включается
3. И так пока не дойдём до последнего фото (т.е. нажали "следующее" 3 раза), после того, как дошли, кнопка "следующее" отключается
4. То же самое в обратном порядке

Бот будет кликать на кнопки, проверять, правильно ли вы прописали атрибуты и отключаются ли кнопки в нужный момент.



# ДЗ №2: Photo Viewer

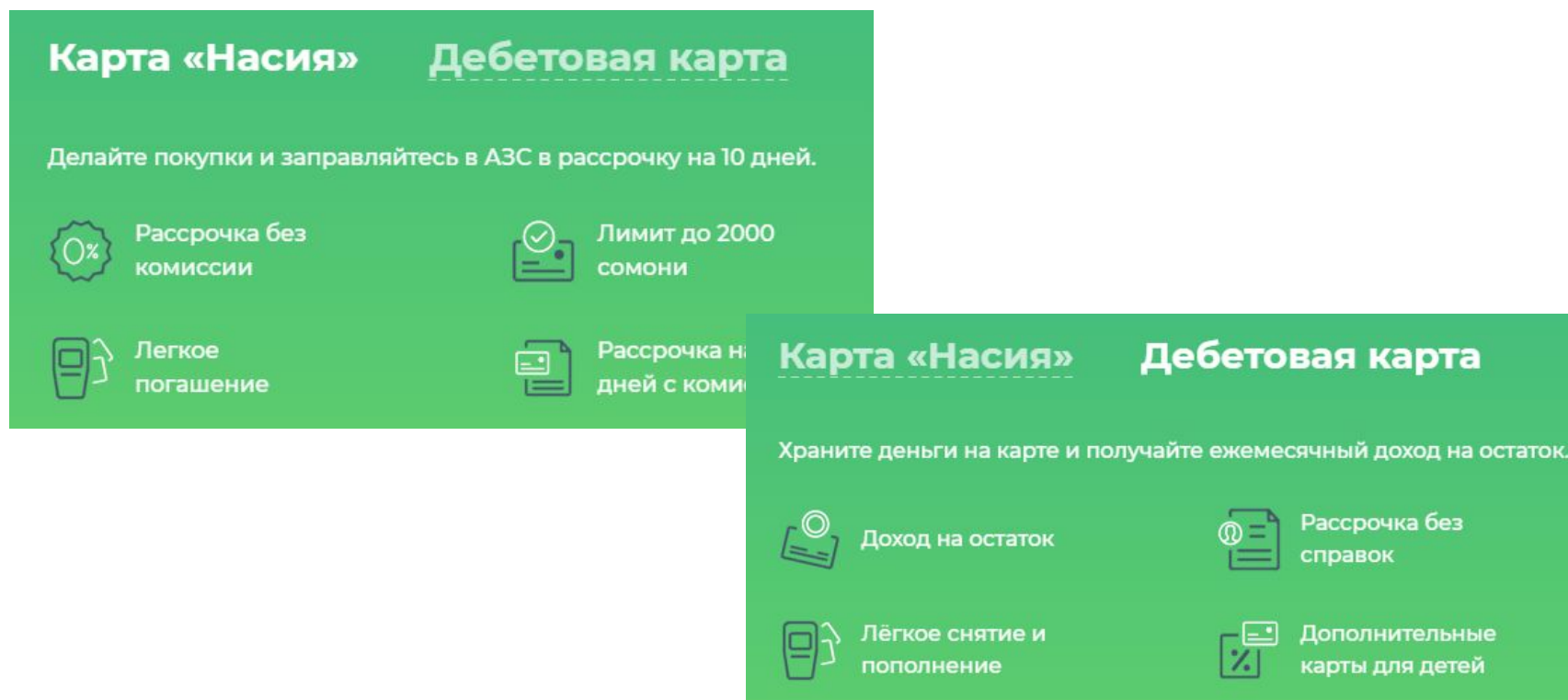
Код в архиве должен располагаться в каталоге [photo-viewer](#).





# ДЗ №3: Насия

Нам поручили сделать виджет с вкладками, который позволяет вам переключать, что показывать:



# ДЗ №3: Насия

Используйте подход с `display="none"` для того, чтобы скрывать и показывать карточку при клике.



# ДЗ №3: Насия

Бота будет устраивать вот такая разметка:

```
11 <body>
12   <div data-id="cards">
13     <div data-id="tabs">
14       <button data-tab="nasia">Карта «Насия»</button>
15       <button data-tab="alifmobi">Дебетовая карта</button>
16     </div>
17     <div data-id="tab-panes">
18       <div data-tabpane="nasia">Делайте покупки и заправляйтесь в АЗС в рассрочку на 10 дней.</div>
19       <div data-tabpane="alifmobi">Храните деньги на карте и получайте ежемесячный доход на остаток.</div>
20     </div>
21   </div>
22
23 </body>
```

Конечно, же не забудьте подключить скрипт.



# ДЗ №3: Насия

Важно:

1. Когда вы делаете официальные сайты, то все буквы, символы и всё остальное должно быть в точности, как вам указали (включая вид кавычек)
2. Вам нужно сделать только виджет, который только переключает вкладки (никаких объектов, которые хранят данные вроде названия карты и текста – не нужно).



# ДЗ №3: Насия

Код в архиве должен располагаться в каталоге [nasia](#).



# ДЗ №4: Cyclic Viewer

Доработайте ДЗ №2 следующим образом: мы хотим, чтобы когда viewer дошёл до последнего фото при нажатии на кнопку "следующее" он показывал первое и продолжал по цепочке. То же самое с кнопкой "предыдущее": когда доходим до первого фото и нажимаем на "предыдущее" должны показывать последнее, предпоследнее и т.д.



# ДЗ №4: Cyclic Viewer

Код в архиве должен располагаться в каталоге [cyclic-viewer](#).



# ДЗ №5: Video Player

Мы хотим научиться делать свой видео-плеер, со своими кнопками управления.

Для отображения видео на странице можно использовать элемент `video`:

```
<body>
  <div data-id="player">
    <video data-id="video" src=""></video>
    <div data-id="controls">
      <button data-action="play">Воспроизведение</button>
      <button data-action="pause">Пауза</button>
      <button data-action="volume-plus">Громче</button>
      <button data-action="volume-minus">Тише</button>
    </div>
  </div>
</body>
```

Конечно же, не забудьте подключить скрипт.

Что мы хотим, мы хотим, чтобы с помощью наших кнопок можно было управлять видео-плеером. Для этого вам нужно будет прочитать статью на MDN или спецификацию HTML.





# ДЗ №5: Video Player

Бот будет кликать на кнопки и проверять, действительно ли они работают и видео воспроизводится.

В качестве видео-ролика предлагаем вам взять вот этот ([весит немного](#)) и положить его в `video/video.mp4`

Архив ДЗ нужно загружать вместе с видео (пропишите путь `video/video.mp4` в `src` элемента `video`).




# ДЗ №5: Video Player

Код в архиве должен располагаться в каталоге [video-player](#).



# ДЗ №6: MonoShop

Представим, что у нас с вами магазин, который торгует только одним видом

Товар		Цена за ед.	Кол-во	Итого
	Nokia 105	239 с.	<div>1<div>+</div><div>-</div></div>	239 с.


Что может делать пользователь:

1. Кликать на кнопки + (максимум до 10) и - (минимум до 1)
2. Когда пользователь кликает на кнопки цена **итого** автоматически пересчитывается



## ДЗ №6: MonoShop

Когда пользователь докликал до минимума (до 1) следующие клики не должны уменьшать количество (кнопки не отключаются), но должно появиться

Товар		Цена за ед.	Кол-во	Итого
	Nokia 105	239 c.	<div>1</div> <div>+</div> <div>-</div>	239 c.


1 шт - минимальный размер заказа

Важно: сообщение должно появляться только тогда, когда пользователь дошёл до 1 и ещё раз кликнул.



# ДЗ №6: MonoShop

Когда пользователь докликал до максимума (10 шт), то дальше не кликается и выводится сообщение:


Товар		Цена за ед.	Кол-во	Итого
	Nokia 105	239 с.	10 <div>+ -</div>	2390 с.

10 шт - максимум в одни руки

Важно: сообщение должно появляться только тогда, когда пользователь дошёл до 10 и ещё раз кликнул.



# ДЗ №6: MonoShop

Товар		Цена за ед.	Кол-во	Итого
	Nokia 105	239 с.	<div>10<div>+</div><div>-</div></div>	2390 с.

10 шт - максимум в одни руки

`data-id="message"`

`data-id="qty"`

`data-action="inc"`

`data-action="dec"`

`data-id="total"`

Бота будут интересовать элементы с `data`-атрибутами, указанными на слайде.



# ДЗ №6: MonoShop

Информация о самом телефоне: id, стоимость за штуку, название и картинка должны храниться в объекте с именем `nokia`.

Информация о заказе (id заказа, id товара, кол-во штук и стоимость за штуку) в объекте с именем `order`.



# ДЗ №6: MonoShop

Код в архиве должен располагаться в каталоге [monoshop](#).





Спасибо за внимание

**alif skills**

2023г.

